2
BACKGROUND AND STATE OF THE ART

You must have a map, no matter how rough. Otherwise you wander all over the place.

— J.R.R. Tolkien

This chapter discusses the state-of-the-art in the areas of WebAssembly and Software Diversification. In Section 2.1 we discuss WebAssembly, focusing on its design and security model. Besides, we discuss the current state-of-the-art of WebAssembly research. In Section 2.2 we discuss related works in the area of Software Diversification. Moreover, we delve into the open challenges regarding the diversification of WebAssembly programs.

2.1 WebAssembly

The W3C publicly announced the WebAssembly (Wasm) language in 2017 as the fourth scripting language supported by all major web browser vendors. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [7]. It is designed to be fast, portable, self-contained, and secure.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Previous works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup time and memory usage than containerization and virtualization [10, 11]. Consequently, in 2019, the Bytecode Alliance proposed WebAssembly System Interface (WASI) [41]. WASI pioneered the execution of WebAssembly with a POSIX system interface protocol, making it possible to execute Wasm closer to the underlying operating system. Therefore, it standardizes the adoption of WebAssembly in heterogeneous platforms [42], i.e., IoT and Edge computing [43, 44].

Currently, WebAssembly serves a variety of functions in browsers, ranging from gaming to cryptomining [45]. Other applications include text processing,

[0] Compilation probe time 2023/12/11 08:51:26

visualization, media processing, programming language testing, online gambling, bar code and QR code fast reading, hashing, and PDF viewing. On the backend, WebAssembly notably excels in short-running tasks. As such, it is particularly suitable for Function-as-a-Service (FaaS) platforms [12] like Cloudflare and Fastly. The subsequent text in this chapter focuses specifically on WebAssembly version 1.0. However, the tools, techniques, and methodologies discussed apply to future WebAssembly versions.

2.1.1 From source code to WebAssembly

WebAssembly programs are compiled from source languages like C/C++, Rust, or Go ahead of time, which means that Wasm binaries can benefit from the optimizations of the source language compiler. The resulting WebAssembly program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code in the browser.

In Listing 2.1 and Listing 2.2, we present a Rust program alongside its corresponding WebAssembly binary. The Rust program in Listing 2.1 iteratively calculates the Fibonacci sequence up to a given number that comes from the host engine. The code in the program encompasses various elements such as vector allocations, external function usage, and a function definition that includes a loop, conditionals, function calls, and memory accesses. The Wasm code shown in Listing 2.2 is simplified in its textual format, known as WAT[1]. The function prototype in lines 4 and 5 of Listing 2.1 are converted into an imported function, as seen in lines 8 and 9 of Listing 2.2. The `fibonacci` function, spanning lines 7 to 20 in Listing 2.1, is compiled into a Wasm function covering lines 14 to 31 in Listing 2.2. Within this function, the translation of various Rust language constructs into Wasm can be observed. For instance, the `for` loop found in line 14 of Listing 2.1 is mapped to a block structure in lines 17 to 31 of Listing 2.2. The breaking condition of the loop is transformed into a conditional branch, as depicted in line 23 of Listing 2.2. In this scenario, the function yields the final set value in the `local` variable. Note that for optimization purposes, the loop concludes by returning the result value, instead of returning post-completion of the loop.

[1]The WAT text format is primarily designed for human readability and for low-level manual editing.

```
...
// Imported form host
extern "C" {
    fn log(s: &str);
    fn get_input() -> usize; }

fn fibonacci(n: usize) -> i32 {
    // Iterative fibonacci
    // Create a vector of size n+1
    let mut fibo_result = vec![0; n + 1];
    // Set ith 0 and 1
    fibo_result[0] = 1;
    fibo_result[1] = 1;
    for i in 2..=n {
        // f[i] = f[i-1] + f[i-2]
        fibo_result[i] = fibo_result[i - 1] + fibo_result[i - 2];
    }
    // Return the last element
    return fibo_result[n];
}
// Pub to export the function
pub fn main() {
    // Get the input from the user
    let ith = get_input();
    // Calculate the fibonacci
    let fib = fibonacci(get_input());
    // Print the result in the host imported function
    log(&format!("{}",fib));
}
```

Listing 2.1: *Example Rust program which includes, external function usage, a function definition featuring a loop, function calls, imported functions, and memory accesses.*

There are several compilers that turn source code into WebAssembly binaries. For example, LLVM compiles to WebAssembly as a backend option since its 7.1.0 release in early 2019[2], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[3]. Significantly, a study by Hilbig [45] reveals that 70% of WebAssembly binaries are generated using LLVM-based compilers. The main advantage of using LLVM is that it provides a modular and state-of-the-art optimization infrastructure for WebAssembly binaries. Today, Emscripten[4] is the most frequently used tool for porting C/C++ code to the Web as a drop-in replacement for a standard compiler like gcc or clang. The main advantage of Emscripten is that it provides a complete toolchain for compiling C/C++ code to WebAssembly, including the automatic generation of the external functions for interacting with a Web host environment. Recently, the Kotlin Multiplatform framework [5] has incorporated WebAssembly as a compilation target, enabling the compilation of Kotlin code to WebAssembly. Similarly, the Cheerp[6] project proposes a Java Virtual Machine(JVM) fully ported to WebAssembly, supporting

[2] https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0
[3] A subset of the TypeScript language
[4] https://emscripten.org/docs/tools_reference/emcc.html
[5] https://kotlinlang.org/docs/wasm-overview.html
[6] https://labs.leaningtech.com/blog/cheerpj-3-deep-dive

Java applications and legacy applets in the browser.

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
  ...
  ; Type section: 0x01 0x00 0x00 0x00 0x13 ... ;
  (type (;type index 0;) (func (param i32 i32)))
  ...
  ; Import section: 0x02  0x00 0x00 0x00 0x57 ... ;
  (import "__wbg__" "__wbg_log" (func (;1;) (type 0)))
  (import "__wbg__" "__wbg_getinput" (func (;2;) (type 8)))
  ...
  ; Custom section: 0x00 0x00 0x00 0x00 0x7E ;
  (@custom "name" "...")
  ...
  (func (;func index 40;) (type 1) (param i32) (result i32)
    (local i32 i32 i32 i32 i32) ;local variables;
    ...
    loop  ; label = @1 ;
      ...
      i32.eqz
      if  ; label = @2 Compare the top of the stack ;
        ...
        local.get 0
        return ; Return the last element which is saved in local 0 ;
      end
      ...
      block  ;label = @2 ;
        ...
        i32.store ; Store the fib value in the mem assigned to the
            ↪ result array;
        br 1 (;@1;) ;Continue the loop;
      end
    end)
  ...
  (func (;44;) (type 8) (result i32)
    ...
    call 2 ; Calling the imported function to get input ;
    i32.store ; Store the input in memory ;
  ...
  (func (;45;) (type 7)
    (local i32 i32 i32)
    ...
    call 44
    call 40 ; Calling fibo function ;
    i32.store offset=20
  ...
  (table (;0;) 33 33 funcref)
  ; Memory section: 0x05 0x00 0x00 0x00 0x03 ... ;
  (memory (;0;) 17)
  ; Global section: 0x06 0x00 0x00 0x00 0x11.. ;
  (global (;global index 0;) (mut i32 ;mut global;) (i32.const 1048576))
  ...
  ; Export section: 0x07 0x00 0x00 0x00 0x72 ... ;
  (export "memory" (memory 0))
  (export "fibo" (func 40))
  (export "main" (func 45))
  ...
  ; Data section: 0x0d 0x00 0x00 0x03 0xEF  ... ;
  (data (;data segment index 0;) (i32.const 1048576) "invalid args...")
  ...
; Custom section: 0x00 0x00 0x00 0x00 0x2F ;
  (@custom "producers" "...")
```

Listing 2.2: *Refer to Listing 2.1 for the Rust code example. This example showcases the transition from Rust to Wasm, where numerous high-level language attributes convert into multiple Wasm instructions. For clarity, we've marked elements and portions of the WebAssembly binary as comments.*

A recent trend in the WebAssembly ecosystem involves porting various programming languages by converting both the language's engine or interpreter and the source code into a WebAssembly program. For example, Javy[7] encapsulates JavaScript code within the QuickJS interpreter, demonstrating that direct source code conversion to WebAssembly isn't always required. If an interpreter for a specific language can be compiled to WebAssembly, it allows for the bundling of both the interpreter and the language into a single, isolated WebAssembly binary. Similarly, Blazor[8] facilitates the execution of .NET Common Intermediate Language (CIL) in WebAssembly binaries for browser-based applications. However, packaging the interpreter and the code in one single standalone WebAssembly binary is still immature and faces challenges. For example, the absence of JIT compilation for the "interpreted" code makes it less suitable for long-running tasks [46, 47]. On the other hand, it proves effective for short-running tasks, particularly those executed in Edge-Cloud computing platforms.

2.1.2 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized, being a consecutive collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by a 4-byte section size, and concludes with the section content, which precisely matches the size indicated earlier. A WebAssembly binary contains sections of 13 types, each with a specific semantic role and placement within the module. For instance, the *Custom Section* stores metadata like the compiler used to generate the binary, while the *Type Section* contains function signatures that serve to validate the *Function Section*. The *Import Section* lists elements imported from the host, and the *Function Section* details the functions defined within the binary. Other sections like *Table, Memory, and Global Sections* specify the structure for indirect calls, unmanaged linear memories, and global variables, respectively. *Export, Start, Element, Code, Data, and Data Count Sections* handle aspects ranging from declaring elements for host engine access to initializing program state, declaring bytecode instructions per function, and initializing linear memory. Each of these sections must occur only once in a binary and can be empty. For clarity, we also annotate sections as comments in the Wasm code in Listing 2.2.

A WebAssembly binary can be processed efficiently due to its organization into a contiguous array of sections. For instance, this structure permits compilers to boost the compilation process either through parallel parsing. Moreover, the *Code Section*'s instructions are further compacted through the use of the LEB128[9] encoding. Consequently, Wasm binaries are not only fast to validate and compile, but also swift to transmit over a network.

[7] `https://github.com/bytecodealliance/javy`
[8] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`
[9] `https://en.wikipedia.org/wiki/LEB128`

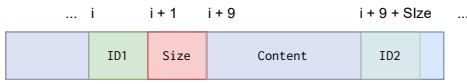


Figure 2.1: *Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.*

2.1.3 WebAssembly's runtime

The WebAssembly's runtime characterizes the behavior of WebAssembly programs during execution. This section describes the main components of the WebAssembly runtime, namely the execution stack, functions, memory model, and execution process. These components are crucial for understanding both the WebAssembly's control flow and the analysis of WebAssembly binaries.

Execution Stack: At runtime, WebAssembly engines instantiate a WebAssembly module. This module is a runtime representation of a loaded and initialized WebAssembly binary described in Section 2.1.2. The primary component of a module instance is its Execution Stack. The Execution Stack stores typed values, labels, and control frames. Labels manage block instruction starts and loop starts. Control frames manage function calls and function returns. Values within the stack can only be static types. These types include `i32` for 32-bit signed integers, `i64` for 64-bit signed integers, `f32` for 32-bit floats, and `f64` for 64-bit floats. Abstract types such as classes, objects, and arrays are not supported natively. Instead, these types are abstracted into primitive types during compilation and stored in linear memory.

Functions: At runtime, WebAssembly functions are closures over the module instance, grouping locals and function bodies. Locals are typed variables that are local to a specific function invocation. A function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the execution stack, writes to the execution stack, reads from the linear memory, writes to the linear memory, reads a global, writes a global or modifies the control flow of the function. Recalling the example WebAssembly binary, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 15 and Line 19 in Listing 2.2. When an instruction reads its operands from the stack, it pushes back the result. Notice that, numeric instructions are annotated with their corresponding type.

Memory model: A WebAssembly module instance incorporates three key types of memory-related components: linear memory, local variables and global variables. These components can either be managed solely by the host engine or shared with the WebAssembly binary itself. This division of responsibility is often categorized as *managed* and *unmanaged* memory [20]. Managed refers to components that are exclusively modified by the host engine at the lowest level,

e.g. when the WebAssembly binary is JITed, while unmanaged components can also be altered through WebAssembly opcodes. First, modules may include a linear memory instance, which is a contiguous array of bytes. This linear memory is accessed using 32-bit integers (`i32`) and is shareable only between the initiating engine and the WebAssembly module instance. Generally, the linear memory is considered to be unmanaged, e.g., line 28 of Listing 2.2 shows an explicit memory access opcode. Second, there are global instances, which are variables accompanied by values and mutability flags (see example in line 49 of Listing 2.2). These globals are managed by the host engine, which controls their allocation and memory placement completely oblivious to the WebAssembly binary scope. They can only be accessed via their declaration index, prohibiting dynamic addressing. Third, local variables are mutable and specific to a given function instance (e.g., line 15 and line 22 in Listing 2.2). They are accessible only through their index relative to the executing function and are part of the data managed by the host engine.

WebAssembly module execution: While a WebAssembly binary could be interpreted, the most practical approach is to JIT compile it into machine code [48]. The main reason is that WebAssembly is optimized and closely aligned with machine code, leading to swift JIT compilation for execution. Browser engines such as V8[10] and SpiderMonkey[11] use this strategy when executing WebAssembly binaries in browser clients. In practice, browsers initially employ a baseline compiler to ensure the rapid availability of incoming WebAssembly binaries. Simultaneously, an optimizing compiler operates in the background. Consequently, the first generated machine code is eventually supplanted by the optimized version. Once JITed, the WebAssembly binary operates within a sandboxed environment, accessing the host environment exclusively through imported functions. This sandboxing follows the Software Fault Isolation(SFI) guarantee, meaning that a WebAssembly program cannot arbitrarily access code or data of its runtime.

WebAssembly standalone engines: While initially intended for browsers, WebAssembly has undergone significant evolution, primarily due to WASI[41]. WASI establishes a standardized POSIX-like interface for interactions between WebAssembly modules and host environments. Compilers can generate WebAssembly binaries that implement WASI, which allows execution in standalone engines. These binaries can then be executed by standalone engines across a variety of environments, including the cloud, servers, and IoT devices [49, 48]. Similarly to browsers, these engines often translate WebAssembly into machine code via JIT compilation, ensuring a sandboxed execution process.

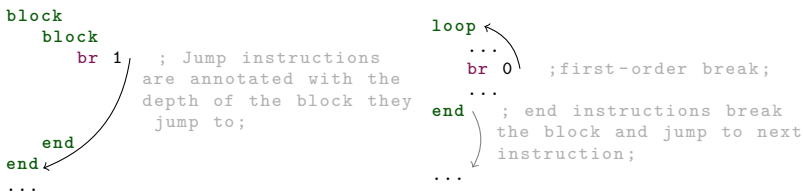
[10] `https://chromium.googlesource.com/v8/v8.git`
[11] `https://spidermonkey.dev/`

Standalone engines such as WASM3[12], Wasmer[13], Wasmtime[14], WAVM[15], and Sledge[50] have been developed to support both WebAssembly and WASI.

2.1.4 WebAssembly's control-flow

A WebAssembly function groups instructions into blocks, with the function's entrypoint acting as the root block. In contrast to conventional assembly code, control-flow structures in Wasm leap between block boundaries rather than arbitrary positions within the code, effectively prohibiting `goto`s to random code positions. Each block may specify the needed execution stack state before execution as well as the resultant execution stack state once its instructions have been executed. Typically, the execution stack state is the quantity and numeric type of values on the stack. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before the execution of the block's instructions. Blocks in Wasm are explicit (see instructions `block` and `end` in lines 16 and 34 of Listing 2.2), delineating where they start and end. By design, a block cannot reference or execute code from external blocks.

During runtime, WebAssembly break instructions can only jump to one of its enclosing blocks. Breaks, except for those within loop constructions, jump to the block's end and continue to the next immediate instruction. For instance, after line 31 of Listing 2.2, the execution would proceed to line 32. Within a loop, the end of a block results in a jump to the block's beginning, thus restarting the loop. For example, if line 29 of Listing 2.2 evaluates as false, the next instruction to be executed in the loop would be line 18. Listing 2.3 provides an example for better understanding, comparing a standard block and a loop block in a Wasm function.



Listing 2.3: *Example of breaking a block and a loop in WebAssembly.*

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the leftmost part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. This design

[12] `https://github.com/wasm3/wasm3`
[13] `https://wasmer.io/`
[14] `https://github.com/bytecodealliance/wasmtime`
[15] `https://github.com/WAVM/WAVM`

hardens the rewriting of WebAssembly binaries. For instance, if an outer block is removed, the depth of the break instructions within nested blocks must be updated to reflect the new enclosing block depth. This is a significant challenge for rewriting tools, as it requires the analysis of the control-flow graph to determine the enclosing block depth for each break instruction.

Notice that, WebAssembly's control flow design adheres to a Control Flow Integrity (CFI) policy. CFI is a security mechanism that limits a program's control flow to a specified set of valid targets, thereby preventing arbitrary jumps [51]. Thus, even when a WebAssembly program originates from potentially untrustworthy sources, CFI policy theoretically guarantees the prevention of arbitrary jumps to random code locations.

2.1.5 Security and reliability for WebAssembly

The WebAssembly ecosystem's expansion needs robust tools to ensure its security and reliability. Numerous tools, employing various strategies to detect vulnerabilities in WebAssembly programs, have been created to meet this need. This section presents a review of the most relevant tools in this field, focusing on those capable of providing security guarantees for WebAssembly binaries.

Static analysis: SecWasm[52] uses information control-flow checking to identify secrecy leaking in WebAssembly binaries. Similarly, Wasmati[53] employs code property graphs for this purpose. Wasp[54] leverages concolic execution to identify potential vulnerabilities in WebAssembly binaries. CT-Wasm[55], verifies the constant time implementation of cryptographic algorithms in WebAssembly. Similarly, Vivienne applies relational symbolic execution to WebAssembly binaries in order to reveal vulnerabilities in cryptographic implementations[56]. While these tools emphasize specific strategies, others adopt a more holistic approach. For example, both Wassail[57] and WasmA[58] provide a comprehensive static analysis framework for WebAssembly binaries. Static analysis tools may have limitations. For instance, a newly, functionally equivalent WebAssembly binary may be generated from the same source code bypassing or breaking the static analysis [59].

Dynamic analysis: Dynamic analysis involves tools such as TaintAssembly[60], which conducts taint analysis on WebAssembly binaries. Furthermore, Stiévenart and colleagues have developed a dynamic approach to slicing WebAssembly programs based on Observational-Based Slicing (ORBS)[61, 62]. This technique aids in debugging, understanding programs, and conducting security analysis. However, Wasabi[63] remains the only general-purpose dynamic analysis tool for WebAssembly binaries, primarily used for profiling, instrumenting, and debugging WebAssembly code. These tools typically analyze software behavior during execution, making them inherently reactive. In other words, they can only

identify vulnerabilities or performance issues while executing input WebAssembly programs.

Protecting WebAssembly binaries and runtimes: The techniques discussed previously are primarily focused on reactive analysis of WebAssembly binaries. However, there exist approaches to harden WebAssembly binaries, enhancing their secure execution, and therefore protecting the security of the entire execution ecosystem. For instance, Swivel[64] proposes a compiler-based strategy designed to eliminate speculative attacks on WebAssembly binaries in Function-as-a-Service (FaaS) platforms by linearizing the machine code from compiling a WebAssembly binary. Similarly, Kolosick and colleagues [65] modify the Lucet compiler to use zero-cost transitions, eliminating the performance overhead of SFI guarantees implementation. In addition, WaVe[66] introduces a mechanized engine for WebAssembly that facilitates differential testing. WaVe can be employed to detect anomalies in engine implementations running Wasm-WASI programs. Much like static and dynamic analysis tools, these tools may suffer from a lack of WebAssembly inputs, which could affect the measurement of their effectiveness.

WebAssembly malware: Since the introduction of WebAssembly, the Web has consistently experienced an increase in cryptomalware. This rise primarily stems from the shift of mining algorithms from CPUs to WebAssembly, a transition driven by notable performance benefits [67]. Tools such as MineSweeper[26], MinerRay[27], and MINOS[28] employ static analysis with machine learning techniques to detect browser-based cryptomalware. In addition, SEISMIC[29], RAPID[30], and OUTGuard[31] leverage dynamic analysis techniques to achieve a similar objective. Moreover, VirusTotal[16], a tool incorporating over 60 commercial antivirus systems as black-boxes, is capable of detecting cryptomalware in WebAssembly binaries. However, obfuscation studies have exposed their shortcomings, revealing an almost unexplored area for WebAssembly that threatens malware detection accuracy. In concrete, Bhansali et al. seminal work[68] demonstrate that cryptomining algorithm's source code can evade previous techniques through the use of obfuscation techniques.

2.1.6 Open challenges

Despite progress in WebAssembly analysis, numerous challenges remain. WebAssembly, though deterministic and well-typed by design, is susceptible to a variety of security threats. First, most existing WebAssembly research is reactive, focusing on detecting and fixing vulnerabilities already reported. This approach leaves WebAssembly binaries and runtime implementations potentially open to unidentified attacks. Second, side-channel attacks present a significant risk. Genkin et al., for example, illustrated how WebAssembly could be manipulated to

[16] `https://www.virustotal.com`

extract data via cache timing-side channels [22]. Furthermore, research conducted by Maisuradze and Rossow demonstrated the potential for speculative execution attacks on WebAssembly binaries [23]. Rokicki et al. disclosed the possibility for port contention side-channel attacks on WebAssembly binaries in browsers [18]. Finally, the binaries themselves may be inherently vulnerable. For example, studies by Lehmann et al. and Stiévenart et al. suggested that flaws in C/C++ source code could infiltrate WebAssembly binaries [20, 21].

2.2 Software diversification

Software diversification involves the synthesis, reuse, distribution, and execution of different, functionally equivalent programs so-called software variants. As outlined in Baudry et al.'s survey [69], Software Diversification falls into five usage categories: reusability [70],performance [71], fault tolerance [72], software testing [73], and security [33]. Our work specifically contributes to the last two categories. Based on the works of Cohen et al. [33], Forrest et al. [34], Jackson et al. [74] and Baudry et al. [69], this section presents core concepts and related works.

Software variants refer to functionally equivalent versions of an original program, produced through Software Diversification at various stages of the software lifecycle, from dependencies (coarse-grained) to machine code levels (fine-grained). The main goal of Software Diversification is to increase the cost of exploitation by making software less predictable. Diversification may be natural [69] or automatic [75]. Natural diversity refers to the side effect of humans creating software variants using different programming languages, compilers, and operating systems [69], all of which adhere to the initial requirements. The software market and competition typically address the creation of natural diversity. For example, Firefox and Chrome web browsers demonstrate natural diversity due to their practical differences in implementation and performance, despite serving the same purpose. This logic extends to operating systems, database engines, virtual machines, and application servers [69]. Natural diversity significantly aids in system security, as different variants are not susceptible to the same vulnerabilities [76, 77]. Unlike N-Version programming [78], natural diversity organically emerges over decades. In other words, while it does not require the allocation of additional human efforts, natural diversity cannot be automatically generated. This is because it is a side effect of the software development process. Given that WebAssembly is a relatively new technology, natural diversity is presently not a feasible option. Hence, for WebAssembly, feasible options are systematic and automatic diversification approaches.

2.2.1 Automatic generation of software variants

The concept of automatic software variants starts with Randell's 1975 work [79], which put forth the notion of artificial fault-tolerant instruction blocks. Artificial Software Diversification, as proposed by Cohen and Forrest in the 1990s [33, 34], gets its development through rewriting strategies. These strategies consist of rule sets for modifying software components to create functionally equivalent, yet distinct, programs. Rewriting strategies typically take the form of tuples: `instr1 => (instr2, instr3, ...)`, where `instr` represents the original code and `(instr2, instr3, ...)` denotes the functionally equivalent code.

Rewriting strategy: The automatic creation of Software Diversification begins with creating rewriting rules. A rewriting rule refers to a functionally equivalent substitution for a code segment, manually written. These rules can be applied at varying levels, from coarse to fine-grained. This can range from the program dependencies level [80] to the instruction level [75]. For example, Cleemput et al. [81] and Homescu et al. [82] inject NOP instructions to yield statically varied versions at the instruction level. Here, the rewriting rule is represented as `instr => (nop instr)`, signifying a `nop` operation preceding the instruction.

Instruction Reordering: This strategy reorders instructions in a program. For example, variable declarations may change if compilers reorder them in the symbol tables. This prevents static examination and analysis of parameters and alters memory locations. In this area, Bhatkar et al. [83, 84] proposed the random permutation of variable and routine order for ELF binaries. Such strategies are not implemented for WebAssembly to the best of our knowledge.

Adding, Changing, Removing Jumps and Calls: This strategy generates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [33] primarily illustrated this concept by inserting random jumps in programs. Pettis and Hansen [85] suggested splitting basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [86] de-inlined basic blocks of code as an LLVM pass. In their approach, each de-inlined code transforms into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [84] extended their previous approach [83], replacing function calls with indirect pointer calls in C source code, allowing post-binary reordering of function calls. In the WebAssembly context, the most analogous work is Wobfuscator [87]. Wobfuscator, a JavaScript obfuscator, substitutes pieces of JavaScript code with WebAssembly code, e.g., numeric calculi. This strategy effectively uses the interleaving of calls between JavaScript and WebAssembly to provide JavaScript variants.

Program Memory and Stack Randomization: This strategy alters the layout of programs in the host memory. Additionally, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [83, 84] proposes

to randomize the base addresses of applications and library memory regions in ELF binaries. Tadesse Aga and Autin [88], and Lee et al. [89] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan et al. [90] suggest separating a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [91] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization. This makes it very challenging for an attacker to ignore the key to inject executable code. This strategy disrupts the predictability of program execution and mitigates certain exploits such as speculative execution. No work has been found that explicitly applies this strategy to WebAssembly.

ISA Randomization and Simulation: This strategy involves using a key to cipher the original program binary into another encoded binary. Once encoded, the program can only be decoded at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [92], and Barrantes et al. [93] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [94] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [95] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Generally, *ISA randomization and simulation* usually faces a performance penalty, especially for WebAssembly, due to the decoding process as shown in WASMixer evaluation [59].

Code obfuscation: Code obfuscation can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as is in any client [96]. Yet, both strategies aim to tackle static reverse engineering of programs. In the context of WebAssembly, Romano et al. [87] proposed an obfuscation technique, wobfuscator, for JavaScript in which part of the code is replaced by calls to complementary WebAssembly functions. Yet, wobfuscator targets JavaScript code, not WebAssembly binaries.

Enumerative synthesis: Enumerative synthesis is a fully automated and systematic approach to generate program variants. It examines all possible programs specific to a given language. The process of enumerative synthesis commences with a piece of input program, typically a basic block. Incrementally, using a defined grammar, it generates all programs of size n. A generated program is then checked for equivalence to the original program, either by using a test suite or a theorem solver. If the generated variant is proven, it is added to the variant's collection. The procedure continues until all potential

programs have been explored. This approach proves especially effective when the solution space is relatively small or can be navigated efficiently. Jacob and colleagues [97] implemented this strategy for x86 programs. They named this technique superdiversification, drawing parallels to superoptimization [98]. Since this strategy fully explores a program's solution space, it contains the aforementioned strategies as special cases. The application of enumerative synthesis to WebAssembly has not been explored.

2.2.2 Equivalence Checking

Equivalence checking between program variants is a vital component for any program transformation task, ranging from checking compiler optimizations [99] to the artificial synthesis of programs discussed in this chapter. It proves that two pieces of code or programs are functionally equivalent [100]. We can roughly simplify the checking process with the following property: two programs are deemed equivalent if they generate identical outputs when given identical inputs from a closed collection of inputs [101]. We adopt this definition of *functional equivalence modulo input* throughout this dissertation. In Software Diversification, equivalence checking seeks to preserve the original functionality of programs while varying observable behaviors. Two programs, for instance, can differ statically and still compute the same result. We outline three methods to check variant equivalence: by construction, check modulo tests and prove-driven equivalence checking.

Equivalence by construction: The equivalence property can be guaranteed by construction. As previously mentioned, Cleemput et al. [81] and Homescu et al. [82] exemplify transformation strategies that generate semantically equivalent program variants. These variants are equivalent by construction. In their case, NOP instructions produce statically different variants. NOP operations, interleaved by any other type of original instruction, serve as a functionally equivalent replacement. However, developer errors may occur during this process, necessitating further validation. The test suite of the original program can serve as a check for the variant.

Checking modulo tests: The process of checking modulo tests involves utilizing a test suite to confirm the equivalence of program variants [102, 103]. When a program variant successfully passes the test suite, it is determined equivalent to the original. It is reasonable to assume that projects prioritizing quality and security are likely to have a robust test suite that facilitates this type of equivalence verification. However, this technique's effectiveness is bounded by the necessity for a preexisting test suite. Yet, as an alternative, fuzzers can be used to automatically generate tests [104]. Fuzzers operate by randomly generating inputs that lead to different observable behaviors. If a variant produces a different output from two identical inputs, it is not equivalent to the original program. Fuzzers' primary drawback is their time-consuming nature and the requirement

for manually introducing oracles. Recent advancements in the field of machine learning have led researchers to explore the application of neural networks in verifying program equivalence. Zhang and his team's work provides an example of this, where Large Language Models are used to generate reference oracles and test cases [105]. Despite its effectiveness, this method attains an accuracy rate of just 88%, which falls short of providing complete verification.

Formal checking: In the absence of a test suite or a technique that inherently implements the equivalence property, the works mentioned earlier use theorem solvers (SMT solvers) [106] to prove the equivalence of program variants. The central idea for SMT solvers is to convert the two code variants into mathematical formulas. The SMT solver then checks for counter-examples [107]. When it finds a counter-example, there is an input for which the two mathematical formulas yield different outputs. The primary limitation of this technique is that not all algorithms can be translated into a mathematical formula, such as loops. Nevertheless, this technique is frequently used for checking no-jump-programs like basic block and peephole replacements [108].

2.2.3 Variants deployment

Program variants, once generated and verified, may be used in two primary scenarios: Randomization or Multivariant Execution (MVE) [74].

Randomization: In the context of our work, the term *Randomization* denotes a program's ability to present different variants to different clients. In this setup, a program, chosen from a collection of variants (referred to as the program's variant pool), is assigned to a random client during each deployment. Jackson et al. [74] define the variant pool in Randomization as herd immunity, as vulnerable binaries can only affect a segment of the client community. El-Khalil and colleagues [109] suggest employing a custom compiler to generate varying binaries from the compilation process. They adapt a version of GCC 4.1 to partition a conventional stack into several component parts, termed multistacks. Similarly, Singhal and colleagues, propose Cornucopia [110]. Cornucopia generates multiple variants of a program by using different compiler flag combinations. Aga and colleagues propose the generation of program variants through the randomization of its data layout in memory[88]. This method allows each variant to operate on the same data in memory but at different memory offsets. Randomization can also be applied to virtual machines and operating systems. On this note, Kc et al. [92] establish a unique mapping between artificial CPU instructions and actual ones, enabling the assignment of various variants to specific target clients. In a similar vein, Xu et al. [91] recompile the Linux Kernel to minimize the exposure time of persistent memory objects, thereby increasing the frequency of address randomization.

Multivariant Execution (MVE): Multiple program variants are composed

into a single binary, known as a multivariant binary [111]. Each multivariant binary is randomly deployed to a client. Then, the multivariant binary executes its embedded program variants at runtime. These embedded variants can either execute in parallel to check for inconsistencies, or as a single program to randomize execution paths [83]. Bruschi and colleagues extend the concept of executing two variants in parallel, introducing non-overlapping and randomized memory layouts [112]. At the same time, Salamat et al. modify a standard library to generate 32-bit Intel variants. These variants have a stack that grows in the opposite direction, allowing for the detection of memory inconsistencies [113]. Davi and colleagues propose Isomeron, an approach for execution-path randomization [114]. Isomeron operates by simultaneously loading the original program and a variant. It then uses a coin flip to determine which copy of the program to execute next at the function call level. Previous works have highlighted the benefits of limiting execution to only two variants in a multivariant environment. Agosta and colleagues, as well as Crane and colleagues, used more than two generated programs in the multivariant composition, thereby randomizing software control flow at runtime [115, 86]. Both strategies have proven effective in enhancing security by addressing known vulnerabilities, such as Just-In-Time Return-Oriented Programming (JIT-ROP) attacks [116] and power side-channel attacks [117]. Lastly, only Voulimeneas et al. [118] have recently proposed a multivariant execution system that enhances security by parallelizing the execution of variants across different machines.

2.2.4 Measuring Software Diversification

Measuring Software Diversification presents a significant challenge. The size of the variant space does not necessarily correlate with a variant's capacity to fulfill an objective such as hardening attacks by making systems less predictable [33]. Ideally, real scenarios would provide the most accurate measurement of diversification, e.g., demonstrating a variant's effectiveness under specific attacks. However, such an approach is not always feasible, i.e., Software Diversification is a preventive strategy. Hence, a combination of static and dynamic metrics is required for measuring Software Diversification.

Static comparison of variants: Static metrics are used to measure the diversity of programs without needing execution. The fundamental concept entails comparing variant source codes or binary codes to determine how diverse they are. Usually, comparing variants means defining a distance metric between programs [101] where the more different the programs are, the greater the distance. At the low-level of bytecode instructions, for example, these metrics include counting instructions [119], Levenshtein distance [120], and global alignments [36]. On the other hand, at the high-level of source code, these metrics often rely on Abstract Syntax Tree (AST) diffing, such as GUMtree-based distances [121] or machine learning inference [122]. As an example of measuring

the diversification, Bostani et al. [123] illustrate the use of static distances in guiding the generation process of variants. They categorize the space of Android applications into malware and goodware. Then, they create malware variants by employing a static distance metric to approach the goodware group as closely as possible, thus successfully evading malware classifiers.

Dynamic comparison of variants: Static comparisons between variants inherently have limitations. For example, two variants may show differences at the source code level but exhibit identical behavior during execution. Take the addition of `nop` operations to a program as an instance. Despite source code level differences, the variant and the original program execute identical instructions, leading to similar behaviors modulo input. Measuring Software Diversification primarily aims to demonstrate variant-specific observabilities. While static differences are observable, runtime information holds complementary relevance [124]. Therefore, dynamic metrics are essential to assess the diversity of variants. For instance, Forrest et al. [125] were pioneers in classifying program behaviors by analyzing their system call traces using n-grams profiling. Cabrera et al. used a global alignments approach to gauge the diversity of JavaScript bytecode traces within the Chrome browser [14]. Fang et al. proposed a method to counteract JavaScript obfuscation techniques used in malicious code, by analyzing dynamic information captured from V8 bytecode traces [126]. Dynamic metrics are primarily employed to cluster similar behaviors. Following the same logic, the diversity is greater when the difference between behaviors is larger. Notice that, dynamic metrics can be difficult due to the expense of program execution or the complication of required user interaction. On the other hand, malware programs, which usually do not require user interaction, are simpler to evaluate in controlled environments before actual deployment.

In the context of WebAssembly, there exist no explicit works on Software Diversification. Consequently, previous metrics have not been directly applied to measure diversification in WebAssembly binaries. However, in other domains, such as the analysis of WebAssembly binaries, several studies have employed static metrics. For example, VeriWasm quantifies attack-based patterns, stating that a WebAssembly binary is more secure with a lower pattern count [127]. This metric might potentially serve as a guide during variant generation. In the field of malware detection, MINOS [28] proposes transforming WebAssembly binaries into grayscale images. They then employ convolutional neural networks to identify malware, where an increased similarity to a malware image increases the probability of the binary being malware. Regarding the dynamic comparisons, Wang et al.'s study [29] profiles WebAssembly instructions during runtime to identify malicious behavior.

2.2.5 Offensive or Defensive assessment of diversification

Lundquist and colleagues [75] distinguish Software Diversification into two categories: Defensive and Offensive Diversification. On the one hand, Defensive Software Diversification introduces unpredictability in system behavior. By making software less predictable, defensive Software Diversification aims to proactively deter attacks, acting as a complementary strategy to other, more reactive, security measures. The majority of previously discussed works in this section contribute to defensive diversification. Yet, Software Diversification that aims to create diverse harmful programs is considered Offensive Diversification [128].

Offensive Diversification: Offensive Diversification is conceptually equal to Defensive Software Diversification. Yet, in an offensive context, one may apply diversification techniques to malware or other malicious codes to evade detection by security software [129]. One might equate Offensive Diversification with Code obfuscation, if its purpose shifts from preventing reverse engineering by malicious actors, to evading detection by malware analysis systems.

Malicious actors may employ previously discussed diversification strategies to evade detection [130]. For instance, in the Web context, Weihang et al. propose to randomly transform HTML elements of web pages to evade advertisement blockers [131]. Over time, evasion techniques have evolved in both complexity and sophistication [132]. Chua et al. [133], for instance, suggested a framework for automatically obfuscating the source code of Android applications using method overloading, opaque predicates, try-catch, and switch statement obfuscation, resulting in multiple versions of identical malware. Moreover, machine learning approaches have been used to develop evasive malware [134], drawing on a corpus of pre-existing malware [123]. These methods aim to thwart static malware detectors, yet, more advanced techniques focus on evading dynamic detection mostly by employing throttling [135, 136].

The term Offensive Software Diversification may seem counterintuitive. Yet, such approaches measure the resilience and accuracy of security systems. This is an almost unexplored area in WebAssembly, posing a threat to malware detection accuracy. Specifically, only Bhansali et al. seminal work[68] has demonstrated that a cryptomining algorithm's source code can evade pre-existing malware detection methods. More recently, Madvex [137] has sought to obfuscate WebAssembly binaries to achieve malware evasion, but this approach is limited to altering only the code section of WebAssembly binaries.

2.3 Open challenges for Software Diversification

As outlined in Section 2.1.6, our primary motivation for the contributions of this thesis is the open issues within the WebAssembly ecosystem. We see potential in employing Software Diversification to address them. Based on our

previous discussion, we highlight several open challenges in the realm of Software Diversification for WebAssembly. First, WebAssembly, being an emerging technology, is in the process of implementing defensive measures. In addition, while measures for WebAssembly can be standardized, the implementation of these standards across the ecosystem is naturally slow. Therefore, applying Software Diversification directly to the generation of WebAssembly binaries, according to any given specification, could serve as a valuable strategy to lessen the impact of vulnerabilities. Second, despite the abundance of related work on software diversity, its exploration in the context of WebAssembly remains limited. This thesis is the first to investigate Software Diversity in depth for the emerging WebAssembly ecosystem. Third, both randomization and multivariant execution remain largely unexplored within the WebAssembly context. The deployment of Software Diversification in WebAssembly poses unique challenges. WebAssembly ecosystems are remarkably dynamic. Web browsers and FaaS platforms serve as prime examples. In these environments, WebAssembly binaries are served millions of times simultaneously to the former, while new WebAssembly binaries are cold-spawned and executed upon each user request in the latter. Thus, designing practical Software Diversification for WebAssembly requires careful consideration of the deployment environment. Last but not least, research on malware detection, as discussed in Section 2.1.5, suggests that offensive diversification may assist in evaluating the resilience and accuracy of WebAssembly's security systems.